Cpp

The "Hello World" program is the first step towards learning any programming language and also one of the simplest programs you will learn. All you have to do is display the message "Hello World" on the screen. Let us now look at the program:

#include<iostream>

Using namespace std;

Int main(){

cout<<”Hello !! World”;

return 0;

}

1. Any line beginning with '//' without quotes OR in between /\*...\*/ in C++ is comment.
2. The **#include** directive tells the compiler to include a file and **#include<iostream>** . It tells the compiler to include the standard iostream file which contains declarations of all the standard input/output library functions.
3. **using namespace std**: This is used to import the entirety of the std namespace into the current namespace of the program. The statement using namespace std is generally considered a bad practice. When we import a namespace we are essentially pulling all type definitions into the current scope. The std namespace is huge. The alternative to this statement is to specify the namespace to which the identifier belongs using the scope operator(::) each time we declare a type.
4. **int main()**: This line is used to declare a function named "main" which returns data of integer type. A function is a group of statements that are designed to perform a specific task. Execution of every C++ program begins with the main() function, no matter where the function is located in the program. So, every C++ program must have a main() function.
5. **{ and }**: The opening braces '{' indicates the beginning of the main function and the closing braces '}' indicates the ending of the main function. Everything between these two comprises the body of the main function
6. **return 0;** : This is also a statement. This statement is used to return a value from a function and indicates the finishing of a function. This statement is basically used in functions to return the results of the operations performed by a function.
7. **Indentation**: As you can see the cout and the return statement have been indented or moved to the right side. This is done to make the code more readable. In a program as Hello World, it does not hold much relevance, but as the programs become more complex, it makes the code more readable, less error-prone. Therefore, you must always use indentations and comments to make the code more readable.

**DATA TYPES**

All [variables](https://www.geeksforgeeks.org/variables-and-keywords-in-c/) use data-type during declaration to restrict the type of data to be stored. Therefore, we can say that data types are used to tell the variables the type of data it can store. Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data-type with which it is declared. Every data type requires a different amount of memory.

Data types in C++ is mainly divided into three types: 

1. **Primitive Data Types**: These data types are built-in or predefined data types and can be used directly by the user to declare variables. example: int, char , float, bool etc. Primitive data types available in C++ are:   
   * Integer
   * Character
   * Boolean
   * Floating Point
   * Double Floating Point
   * Valueless or Void
   * Wide Character
2. [**Derived Data Types:**](https://www.geeksforgeeks.org/derived-data-types-in-c/) The data-types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:   
   * Function
   * Array
   * Pointer
   * Reference
3. [**Abstract or User-Defined Data Types**](https://www.geeksforgeeks.org/user-defined-derived-data-types-in-c/): These data types are defined by user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined datatypes:   
   * Class
   * Structure
   * Union
   * Enumeration
   * Typedef defined DataType
4. **Integer**: Keyword used for integer data types is **int**. Integers typically requires 4 bytes of memory space and ranges from -2147483648 to 2147483647.
5. **Character**: Character data type is used for storing characters. Keyword used for character data type is **char**. Characters typically requires 1 byte of memory space and ranges from -128 to 127 or 0 to 255.
6. **Boolean**: Boolean data type is used for storing boolean or logical values. A boolean variable can store either *true*or *false*. Keyword used for boolean data type is **bool**.
7. **Floating Point**: Floating Point data type is used for storing single precision floating point values or decimal values. Keyword used for floating point data type is **float**. Float variables typically requires 4 byte of memory space.
8. **Double Floating Point**: Double Floating Point data type is used for storing double precision floating point values or decimal values. Keyword used for double floating point data type is **double**. Double variables typically requires 8 byte of memory space.
9. **void**: Void means without any value. void datatype represents a valueless entity. Void data type is used for those function which does not returns a value.
10. [**Wide Character**](https://www.geeksforgeeks.org/wide-char-and-library-functions-in-c/): Wide character data type is also a character data type but this data type has size greater than the normal 8-bit datatype. Represented by **wchar\_t**. It is generally 2 or 4 bytes long.

**Datatype Modifiers**  
As the name implies, datatype modifiers are used with the built-in data types to modify the length of data that a particular data type can hold.

Data type modifiers available in C++ are: 

* **Signed**
* **Unsigned**
* **Short**
* **Long**

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Size (in bytes)** | **Range** |
| **short int** | **2** | **-32,768 to 32,767** |
| **unsigned short int** | **2** | **0 to 65,535** |
| **unsigned int** | **4** | **0 to 4,294,967,295** |
| **int** | **4** | **-2,147,483,648 to 2,147,483,647** |
| **long int** | **4** | **-2,147,483,648 to 2,147,483,647** |
| **unsigned long int** | **8** | **0 to 4,294,967,295** |
| **long long int** | **8** | **-(2^63) to (2^63)-1** |
| **unsigned long long int** | **8** | **0 to 18,446,744,073,709,551,615** |
| **signed char** | **1** | **-128 to 127** |
| **unsigned char** | **1** | **0 to 255** |
| **float** | **4** |  |
| **double** | **8** |  |
| **long double** | **12** |  |
| **wchar\_t** | **2 or 4** | **1 wide character** |

**Note**: Above values may vary from compiler to compiler. In the above example, we have considered GCC 32 bit.  
We can display the size of all the data types by using the sizeof() operator and passing the keyword of the datatype as argument to this function

// C++ program to sizes of data types

#include<iostream>

using namespace std;

int main()

{

cout << "Size of char : " << sizeof(char)

<< " byte" << endl;

cout << "Size of int : " << sizeof(int)

<< " bytes" << endl;

cout << "Size of short int : " << sizeof(short int)

<< " bytes" << endl;

cout << "Size of long int : " << sizeof(long int)

<< " bytes" << endl;

cout << "Size of signed long int : " << sizeof(signed long int)

<< " bytes" << endl;

cout << "Size of unsigned long int : " << sizeof(unsigned long int)

<< " bytes" << endl;

cout << "Size of float : " << sizeof(float)

<< " bytes" <<endl;

cout << "Size of double : " << sizeof(double)

<< " bytes" << endl;

cout << "Size of wchar\_t : " << sizeof(wchar\_t)

<< " bytes" <<endl;

return 0;

}

OUTPUT

Size of char : 1 byte

Size of int : 4 bytes

Size of short int : 2 bytes

Size of long int : 8 bytes

Size of signed long int : 8 bytes

Size of unsigned long int : 8 bytes

Size of float : 4 bytes

Size of double : 8 bytes

Size of wchar\_t : 4 bytes

INPUTS AND OUTPUTS

C++ comes with libraries that provide us with many ways for performing input and output. In C++ input and output are performed in the form of a sequence of bytes or more commonly known as **streams**.

* **Input Stream:** If the direction of flow of bytes is from the device(for example, Keyboard) to the main memory then this process is called input.
* **Output Stream:** If the direction of flow of bytes is opposite, i.e. from main memory to device( display screen ) then this process is called output.

**Header files available in C++ for Input/Output operations are:**

1. **iostream**: iostream stands for standard input-output stream. This header file contains definitions of objects like cin, cout, cerr, etc.
2. **iomanip**: iomanip stands for input-output manipulators. The methods declared in these files are used for manipulating streams. This file contains definitions of setw, setprecision, etc.
3. **fstream**: This header file mainly describes the file stream. This header file is used to handle the data being read from a file as input or data being written into the file as output.

* **Un-buffered standard error stream (cerr)**: The C++ cerr is the standard error stream that is used to output the errors. As cerr in C++ is un-buffered so it is used when one needs to display the error message immediately. It does not have any buffer to store the error message and display it later.
* The main difference between cerr and cout comes when you would like to redirect output using "cout" that gets redirected to file if you use "cerr" the error doesn't get stored in file
* **buffered standard error stream (clog)**: This is also an instance of ostream class and used to display errors but unlike cerr the error is first inserted into a buffer and is stored in the buffer until it is not fully filled. or the buffer is not explicitly flushed (using flush()). The error message will be displayed on the screen too.

<bits/stdc++.h>

It is basically a header file that includes every standard library. In programming contests, using this file is a good idea, when you want to reduce the time wasted in doing chores; especially when your rank is time sensitive.

NAMESPACE IN C++

// A program to demonstrate need of namespace

int main()

{

int value;

value = 0;

double value; // Error here

value = 0.0;

}

Output :

Compiler Error:

'value' has a previous declaration as 'int value'

In each scope, a name can only represent one entity. So, there cannot be two variables with the same name in the same scope. Using namespaces, we can create two variables or member functions having the same name.

Namespaces allow us to group named entities that otherwise would have *global scope* into narrower scopes, giving them *namespace scope*. This allows organizing the elements of programs into different logical scopes referred to by names.

* Namespace is a feature added in C++ and not present in C.
* A namespace is a declarative region that provides a scope to the identifiers (names of the types, function, variables etc) inside it.
* Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.
* Namespace declarations appear only at global scope.
* Namespace declarations can be nested within another namespace.
* Namespace declarations don’t have access specifiers. (Public or private)
* No need to give semicolon after the closing brace of definition of namespace.
* We can split the definition of namespace over several units.

ENDL VS “\N”

Although they both seem to do the same thing, there is a subtle difference between them.

**cout << endl** : Inserts a new line and flushes the stream

**cout << "\n"** : Only inserts a new line.

Therefore,   
cout << endl;   
can be said equivalent to   
cout << '\n' << flush;   
So cout << "\n" seems performance wise better than cout << endl; unless flushing of stream is required.   
Some other differences between endl and \n are:

1. endl is manipulator while \n is character.
2. endl doesn't occupy any memory whereas \n is character so It occupy 1 byte memory.
3. \n being a character can be stored in a string(will still convey its specific meaning of line break) while endl is a keyword and would not specify any meaning when stored in a string.
4. We cannot write endl in between double quotation while we can write \n in between double quotation like   
   cout<<"\n"; it is right but cout<<"endl"; is wrong.
5. We can use \n both in C and C++ but, endl is only supported by C++ and not the C language.

[What happen when we exceed valid range of built-in data types in C++?](https://practice.geeksforgeeks.org/tracks/module-1-basics-and-control-structures/?batchId=113#trackTitle_812_7)

int main()

{

for (char a = 0; a <= 225; a++)

cout << a;

return 0;

}

a is declared as char. Here the loop is working from 0 to 225. So, it should print from 0 to 225, then stop. But it will generate a infinite loop. The reason for this is the valid range of character datatype is -128 to 127. When ‘a’ become 128 through a++, the range is exceeded and as a result the first number from negative side of the range (i.e. -128) gets assigned to a. As a result of this 'a' will never reach at point 225. so it will print the infinite series of character.

{

// declaring Boolean

// variable with true value

bool a = true;

for (a = 1; a <= 5; a++)

cout << a;

return 0;

}

This code will print '1' infinite time because here 'a' is declared as 'bool' and it's valid range is 0 to 1. And for a Boolean variable anything else than 0 is 1 (or true). When 'a' tries to become 2 (through a++), 1 gets assigned to 'a'. The condition a<=5 is satisfied and the control remains with in the loop.

[Pre-increment (or pre-decrement) in C++](https://practice.geeksforgeeks.org/tracks/module-1-basics-and-control-structures/?batchId=113#trackTitle_812_10)

n C++, pre-increment (or pre-decrement) can be used as [l-value](https://www.geeksforgeeks.org/lvalue-and-rvalue-in-c-language/)

int a = 10;

++a = 20; // works

printf("a = %d", a);

WHEREAS,

int a = 10;

a++ = 20; // error

printf("a = %d", a);

[How to use getline() in C++ when there are blank lines in input?](https://practice.geeksforgeeks.org/tracks/module-1-basics-and-control-structures/?batchId=113#trackTitle_812_11)

In C++, if we need to read few sentences from a stream, the generally preferred way is to use getline() function. It can read till it encounters newline or sees a delimiter provided by user.

int main()

{

string str;

int t = 4;

while (t--)

{

// Read a line from standard input in str

getline(cin, str);

cout << str << " : newline" << endl;

}

return 0;}

[Precision of floating point numbers in C++ (floor(), ceil(), trunc(), round() and setprecision())](https://practice.geeksforgeeks.org/tracks/module-1-basics-and-control-structures/?batchId=113#trackTitle_812_12)

Decimal equivalent of 1/3 is 0.33333333333333…. An infinite length number would require infinite memory to store, and we typically have 4 or 8 bytes. Therefore, Floating point numbers store only a certain number of significant digits, and the rest are lost. The **precision** of a floating point number defines how many significant digits it can represent without information loss. When outputting floating point numbers, cout has a default precision of 6 and it truncates anything after that. **ceil():**

Ceil rounds off the given value to the closest integer which is more than the given value.

**trunc():**

Trunc rounds removes digits after decimal point.

**round():**

Rounds given number to the closest integer.

**setprecision():**

Setprecision when used along with 'fixed' provides precision to floating point numbers correct to decimal numbers mentioned in the brackets of the setprecison.

cout << fixed << setprecision(0) << pi <<" "<<npi<<endl;

[Loops in C and C++](https://practice.geeksforgeeks.org/tracks/module-1-basics-and-control-structures/?batchId=113#trackTitle_812_13)

Loops in programming come into use when we need to repeatedly execute a block of statements.

In computer programming, a loop is a sequence of instructions that is repeated until a certain condition is reached.

* An operation is done, such as getting an item of data and changing it, and then some condition is checked such as whether a counter has reached a prescribed number.
* **Counter not Reached:**If the counter has not reached the desired number, the next instruction in the sequence returns to the first instruction in the sequence and repeat it.
* **Counter reached:** If the condition has been reached, the next instruction "falls through" to the next sequential instruction or branches outside the loop.

There are mainly two types of loops:

1. **Entry Controlled loops**: In this type of loops the test condition is tested before entering the loop body. **For Loop** and **While Loop** are entry controlled loops.
2. **Exit Controlled Loops**: In this type of loops the test condition is tested or evaluated at the end of loop body. Therefore, the loop body will execute atleast once, irrespective of whether the test condition is true or false. **do - while loop** is exit controlled loop.

* **Initialization Expression**: In this expression we have to initialize the loop counter to some value. for example: int i=1;
* **Test Expression**: In this expression we have to test the condition. If the condition evaluates to true then we will execute the body of loop and go to update expression otherwise we will exit from the for loop. For example: i <= 10;
* **Update Expression**: After executing loop body this expression increments/decrements the loop variable by some value. for example: i++;

**What about an Infinite Loop?**

An infinite loop (sometimes called an endless loop ) is a piece of coding that lacks a functional exit so that it repeats indefinitely. An infinite loop occurs when a condition always evaluates to true. Usually, this is an error.

**int i;**

// This is an infinite for loop as the condition

// expression is blank

for ( ; ; )

{

printf("This loop will run forever.\n");

}

// This is an infinite for loop as the condition

// given in while loop will keep repeating infinitely

/\*

while (i != 0)

{

i-- ;

printf( "This loop will run forever.\n");

}

\*/

// This is an infinite for loop as the condition

// given in while loop is "true"

/\*

while (true)

{

printf( "This loop will run forever.\n");

}

\*/

[Decision Making in C / C++ (if , if..else, Nested if, if-else-if )](https://practice.geeksforgeeks.org/tracks/module-1-basics-and-control-structures/?batchId=113#trackTitle_812_15)

There come situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code. For example, in C if x occurs then execute y else execute z. There can also be multiple conditions like in C if x occurs then execute p, else if condition y occurs execute q, else execute r. This condition of C else-if is one of the many ways of importing multiple conditions.

Decision-making statements in programming languages decide the direction of the flow of program execution. Decision-making statements available in C or C++ are: 

1. [if statement](https://practice.geeksforgeeks.org/tracks/module-1-basics-and-control-structures/?batchId=113#if)
2. [if..else statements](https://practice.geeksforgeeks.org/tracks/module-1-basics-and-control-structures/?batchId=113#if-else)
3. [nested if statements](https://practice.geeksforgeeks.org/tracks/module-1-basics-and-control-structures/?batchId=113#nested-if)
4. [if-else-if ladder](https://practice.geeksforgeeks.org/tracks/module-1-basics-and-control-structures/?batchId=113#if-else-if)
5. [switch statements](https://www.geeksforgeeks.org/switch-statement-cc/)
6. [Jump Statements:](https://practice.geeksforgeeks.org/tracks/module-1-basics-and-control-structures/?batchId=113#jmp)   
   1. [break](https://practice.geeksforgeeks.org/tracks/module-1-basics-and-control-structures/?batchId=113#brk)
   2. [continue](https://practice.geeksforgeeks.org/tracks/module-1-basics-and-control-structures/?batchId=113#cont)
   3. [goto](https://practice.geeksforgeeks.org/tracks/module-1-basics-and-control-structures/?batchId=113#got)
   4. [return](https://practice.geeksforgeeks.org/tracks/module-1-basics-and-control-structures/?batchId=113#ret)

#### [Switch Statement in C/C++](https://practice.geeksforgeeks.org/tracks/module-1-basics-and-control-structures/?batchId=113#trackTitle_812_16)

Switch case statements are a substitute for long if statements that compare a variable to several integral values

* The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.
* Switch is a control statement that allows a value to change control of execution.

**Syntax:**

switch (n)

{

case 1: // code to be executed if n = 1;

break;

case 2: // code to be executed if n = 2;

break;

default: // code to be executed if n doesn't match any cases

}

**Macros**: Macros are a piece of code in a program which is given some name. Whenever this name is encountered by the compiler the compiler replaces the name with the actual piece of code. The ‘#define’ directive is used to define a macro

In C++, when an integer value is compared with an unsigned it, the int is promoted to unsigned. Negative numbers are stored in 2’s complement form and unsigned value of the 2’s complement form is much higher than the sizeof int.

#include <iostream>  
using namespace std;  
  
int main()  
{  
 if (sizeof(int) > -1)  
 cout << "Yes";  
 else  
 cout << "No";  
 return 0;  
}

Ans no;

How many indicators are available in C++?

Explanation: There are **three indicators** are available in C++. They are Error indicator, End-Of-File indicator and Position indicator.

We are not allowed to do addition operation on cin.

#include < iostream >  
using namespace std;  
  
int main ()  
{  
 int i;  
 cout << "Please enter an integer value: ";  
 cin >> i + 4;  
  
 return 0;  
}

Error;

How many groups of output of operation are there in C++?

Explanation: There are **two groups** of output operation in c++. They are formatted output and unformatted output.

[Arrays in C/C++](https://practice.geeksforgeeks.org/tracks/module-2-arrays-and-strings/?batchId=113#trackTitle_820_1)

An array in C/C++ or be it in any programming language is a collection of similar data items stored at contiguous memory locations and elements can be accessed randomly using indices of an array.  They can be used to store collection of primitive data types such as int, float, double, char, etc of any particular type. To add to it, an array in C/C++ can store derived data types such as the structures, pointers etc

**Why do we need arrays?**   
We can use normal variables (v1, v2, v3, ..) when we have a small number of objects, but if we want to store a large number of instances, it becomes difficult to manage them with normal variables. The idea of an array is to represent many instances in one variable.

**Advantages of an Array in C/C++:**

1. Random access of elements using array index.
2. Use of less line of code as it creates a single array of multiple elements.
3. Easy access to all the elements.
4. Traversal through the array becomes easy using a single loop.
5. Sorting becomes easy as it can be accomplished by writing less line of code.

**Disadvantages of an Array in C/C++:**

1. Allows a fixed number of elements to be entered which is decided at the time of declaration. Unlike a linked list, an array in C is not dynamic.
2. Insertion and deletion of elements can be costly since the elements are needed to be managed in accordance with the new memory allocation.

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array). The base value is index 0 and the difference between the two indexes is the offset.

Arrays vs pointer

Pointers are used for storing address of dynamically allocated arrays and for arrays which are passed as arguments to functions.

1. *Array name gives address of first element of array.*

*Array members are accessed using pointer arithmetic.*   
Compiler uses pointer arithmetic to access array element.

*Array parameters are always passed as pointers, even when we use square bracket.*

being:

1) the sizeof operator  
o sizeof(array) returns the amount of memory used by all elements in array  
o sizeof(pointer) only returns the amount of memory used by the pointer variable itself

2) the & operator  
o &array is an alias for &array[0] and returns the address of the first element in array  
o &pointer returns the address of pointer

3) a string literal initialization of a character array  
o char array[] = “abc” sets the first four elements in array to ‘a’, ‘b’, ‘c’, and ‘\0’  
o char \*pointer = “abc” sets pointer to the address of the “abc” string (which may be stored in read-only memory and thus unchangeable)

4) Pointer variable can be assigned a value whereas array variable cannot be.

**What is vector in C++?**   
A vector in C++ is a class in STL that represents an array. The advantages of vector over normal arrays are,

* We do not need pass size as an extra parameter when we declare a vector i.e, Vectors support dynamic sizes (we do not have to initially specify size of a vector). We can also resize a vector.
* Vectors have many in-built function like, removing an element, etc.

S[td::string class in C++](https://practice.geeksforgeeks.org/tracks/module-2-arrays-and-strings/?batchId=113#trackTitle_820_10)

C++ has in its definition a way to represent **sequence of characters as an object of class**. This class is called std:: string. String class stores the characters as a sequence of bytes with a functionality of allowing **access to single byte character**.

* A character array is simply an **array of characters** can terminated by a null character. A string is a **class which defines objects** that be represented as stream of characters.

Implementation of**character array is faster** than std:: string. **Strings are slower** when compared to implementation than character array.

**Capacity Functions**  
**4. capacity()** :- This function **returns the capacity** allocated to the string, which can be **equal to or more than the size** of the string. Additional space is allocated so that when the new characters are added to the string, the **operations can be done efficiently**.  
**5. resize()** :- This function **changes the size of string**, the size can be increased or decreased.  
**6.length()**:-This function **finds the length of the string**  
**7.shrink\_to\_fit()** :- This function**decreases the capacity**of the string and makes it equal to the minimum capacity of the string. This operation is**useful to save additional memory**if we are sure that no further addition of characters have to be made.

**Iterator Functions**  
**8. begin()** :- This function returns an **iterator** to **beginning** of the string.  
**9. end()** :- This function returns an **iterator** to **end** of the string.  
**10. rbegin()** :- This function returns a**reverse iterator** pointing at the**end**of string.  
**11. rend()** :- This function returns a **reverse iterator** pointing at **beginning** of string.

**Manipulating Functions**  
**12. copy("char array", len, pos)** :- This function**copies the substring in target character array** mentioned in its arguments. It takes 3 arguments,**target char array, length to be copied and starting position in string to start copying.**  
**13. swap()** :- This function **swaps**one string with other.